

Slice-Based Facility Architecture

Larry Peterson, Princeton

Soner Sevinc, Princeton

Jay Lepreau, Utah

Robert Ricci, Utah

John Wroclawski, USC/ISI

Ted Faber, USC/ISI

Stephen Schwab, Sparta

Draft Version 0.8

November 5, 2007

This work is supported in part by NSF grants CNS-0540815 and CNS-0631422.

Table of Contents

1	Introduction.....	3
2	Principals.....	3
3	Abstractions.....	5
3.1	Components.....	5
3.2	Slices.....	6
4	Authorization and Access Control.....	7
4.1	Slices.....	7
4.2	Components.....	8
4.3	Flow.....	8
5	Names & Identifiers.....	9
6	Data Types.....	9
6.1	RSpec.....	9
6.2	Ticket.....	10
6.3	Slice Credential.....	11
7	Interfaces.....	11
7.1	Slice Interface.....	11
7.2	Component Interface.....	12
7.2.1	Embedding a Slice.....	13
7.2.2	Controlling a Slice.....	13
7.2.3	Slice Information.....	14
7.3	Management Interface.....	14
8	Name Registry.....	15
8.1	Example Registries.....	15
8.2	Registry Interface.....	16
9	Implementation Options.....	17

1 Introduction

This document defines a minimal set of interfaces and data types intended to support *slice-based network facilities*. The specification is designed to support facilities like PlanetLab, VINI, and GENI – and assumes the reader is familiar with those systems – but is ultimately intended to support a much broader range of designs than those systems embody.

Although this effort grew out the GENI Initiative, it does not currently have any official standing with GENI. Lacking any such sponsorship – and hoping to foster much broader acceptance – we refer to the architecture defined in this document as the *Slice-based Facility Architecture* (SFA).

SFA’s design is driven by a number of high-level goals. Ultimately, each of these goals, and the architectural decisions and elements provided in support, will be discussed in a separate, topical document. Here, and throughout this document, we touch on each goal as needed to motivate or explain a design decision. Among the key goals driving the SFA design are:

- A well-defined *authentication and authorization* architecture and model.
- Support for *federation* – the composition of coherent larger systems from subsystems with independent, distributed ownership and management policies.
- A clear path to *evolvability and extensibility* of the overall system, through carefully chosen design invariants and clear separation of concerns.

[Issue: discuss Fail-safety.]

[TODO: Roadmap and other related (vertical) documents.]

2 Principals

The SFA recognizes three key actors:

- *Owners* of parts of the network substrate, who are therefore responsible for the externally visible behavior of their equipment, and who establish the high-level policies for how their portion of the substrate is utilized.
- *Operators* of parts of the network, often working for owners, whose job it is to keep the platform running, provide a service to researchers, and prevent malicious or otherwise damaging activity exploiting the platform.
- *Users* (researchers and developers) employing the facility in their work, for running experiments, deploying experimental services, measuring aspects of the platform, and so on.

The SFA must mediate the following activities:

- Allow owners to declare resource allocation and usage policies for substrate facilities under their control, and to provide mechanisms for enforcing those policies. The assumption is that there will be multiple owners, each setting policies for their portion of the overall facility and it will be a *federation* of these facilities and policies that will form the entirety of the network.
- Allow operators to manage the network substrate, which includes installing new physical plant and retiring old or faulty plant, installing and updating system software, and monitoring the network for performance, functionality, and security. Management is likely to be decentralized: there will be more than one organization administering disjoint collections of sites.
- Allow users to create and populate slices, allocate resources to them, and run experiment-specific software in them. Much of this functionality, such as convenient installation of software, including libraries or language runtimes, may be provided by higher-level services; the SFA aims to support the deployment and configuration of such services.

To this end, the SFA defines three principals:

- A *management authority* (MA) is responsible for some subset of substrate components: providing operational stability for those components, ensuring the components behave according to acceptable use policies, and executing the resource allocation wishes of the component owner.
- A *slice authority* (SA) is responsible for the behavior of a set of slices, vouching for the users running experiments in each slice and taking appropriate action should the slice misbehave.
- A *user* is a researcher that wishes to run an experiment or service in a slice, or a developer that provides a service used by other researchers.

[Issue: In this draft of the document, management authorities sometimes conflate the roles of owners and operators. There is some disagreement about the merits of this perspective. The issue revolves around the question of how closely related and scoped the policy determination role of the owner and the operations and management role of the operator are.

In some cases, an MA will correspond to a single organization (e.g., a site), in which case the owner and operator are the same. In other cases, the owner and operator are distinct, with the former establishing a "management agreement" with the latter. One view is that in this case the owner will delegate policy enforcement, as well as O&M, to the operator. The other view is that owners will prefer to delegate O&M to an operator, but retain the policy management role for themselves.

Examples of both are present in the industry today. A contract maintenance or field service organization is an example of the second case – the owner of a computing facility may contract out the task of operating and maintaining the facility, but retain full control of the facility's usage and management. On the other hand, "service based computing" or facility management outsourcing is an example of the first case – one organization may contract with another to

operate and manage a computing facility based only on top-level policy guidelines provided by the originating organization.

Moving to the design of SFA, one perspective is that even in the case where the owner and the operator are the same entity, the roles should be clearly distinguished, with the MA and its operations “modeling” or supporting only the role of the operator. Another perspective is that parsimony of mechanism might argue for the MA to implement aspects of both the “owner” and “operator” role, either always, or in the case where the two roles are lodged in the same entity either directly or by delegation.

In any case, the wording of the document needs to tease these issues apart carefully and present the underlying design decisions and motivations with clarity.]

An additional entity in the system is the *end-user* (or *client*) of the services deployed in slices – an “actual user” of an experimental system operating on the facility, as distinguished from the researcher or developer of such an experiment, modeled here as a User. This report offers no guidance on how these end-user individuals interact with the system, as this is a slice-specific concern.

3 Abstractions

The SFA defines two key abstractions: *components* and *slices*.

3.1 Components

The *Component* is a logical abstraction representing the building blocks of the physical substrate. For example, a component might correspond to an edge computer, a customizable router, or a programmable access point.

A component encapsulates a collection of *resources*, including physical resources (e.g., CPU, memory, disk, bandwidth) logical resources (e.g., file descriptors, port numbers), and synthetic resources (e.g., packet forwarding fast paths). These resources can be contained in a single physical device or distributed across a set of devices, depending on the nature of the component. A given resource can belong to at most one component.

Each component is controlled via a *component manager* (CM), which exports a well-defined, remotely accessible interface. The component manager defines the operations available to higher-level services to manage the allocation of component resources to different users and their experiments.

Each component is assumed to have an owner, who establishes policies about how the component's resources are assigned to users. These policies are enforced by elements of the SFA, as described below.

It must be possible to multiplex (slice) component resources among multiple users. This is normally done by some combination of virtualizing the component (where each user acquires a virtual copy of the component's resources), and by partitioning the component into distinct resource sets (where each user acquires a physical partition of the component's resources). In both cases, we say the user is granted a *sliver* of the component.

The architecture does not, however, assume that a component is infinitely sliceable. Rather, the resources available to users are assumed to be limited, and a central function of SFA is the management of these limited resources. A corollary of this assumption is that the degenerate case of a component that can support only one sliver at a time, representing the entire physical device, is supported.

Each component must include hardware or software mechanisms that isolate slivers from each other, making it appropriate to view a sliver as a “resource container.” However, the SFA architecture itself imposes no requirements on the “quality” of this isolation. The level of isolation provided by a component, and the sufficiency of this isolation for specific experiments and users, is a matter between individual components and users.

A sliver that includes resources capable of loading and executing user-provided programs can also be viewed as supporting an execution environment. Slivers that support such execution environments are said to be active slivers. Other (non-active) slivers might correspond to non-programmable resources; e.g., a fixed-function hardware device, tunnel, VLAN, circuit, or light-path. One distinction between active and non-active slivers is that the latter do not support interfaces concerned with management of user-defined state.

3.2 Slices

From a researcher's perspective, a *slice* is a substrate-wide collection of computing and communication resources capable of running an experiment or a wide-area network service. From an operator's perspective, slices are the primary abstraction for accounting and accountability – resources are acquired and consumed by slices, and external program behavior is traceable to a slice, respectively.

A slice is defined by a set of slivers spanning a set of components, plus an associated set of users that are allowed to access those slivers for the purpose of running an experiment on the substrate. That is, a slice has a name, which is bound to a set of users associated with the slice and a (possibly empty) set of slivers. Both the set of users associated with a slice and the set of components and resources available to the slice may change during the slice's lifetime.

There are three unique stages in the lifetime of a slice, each corresponding to an action (operation) that can be performed on a slice:

- **Create:** the slice exists in name only and is bound to a set of users;
- **Embed:** the slice is instantiated on a set of components and resources assigned to it;
- **Access:** the slice is accessed by, and runs code on behalf of, a user.

A slice has to be created and bound to a set of users before it can be embedded, and it must be embedded before being accessed.

Slices are created (registered) in the context of a slice authority – a principal that takes responsibility for the behavior of the slice. A slice is created only once, but the set of users bound to it can change over time. A slice registration has a finite lifetime; the responsible slice authority must refresh this registration periodically.

Embedding a slice effectively configures the slice on a set of components; this step can be repeated multiple times. In fact, embedding often involves two sub-steps: a slice is first instantiated on a set of components with nil or best-effort resources assigned to it, and later provisioned with additional (perhaps guaranteed) resources, for example, for the duration of a single experiment.

An experiment or service then “runs in” a slice. The usage of the slice’s resources is not defined by the architecture. Once allocated, resources may be configured and controlled in any fashion permitted by the (owners of) components supplying those resources. Multiple experiments can be run in a single slice, sequentially or in parallel. For each run, the experiment may change parameters but leave the slice configuration (embedding) unchanged, or it may change either the set of components or the resources allocated on those components, or both. How rapidly a slice can be reconfigured to support a new experiment depends on the implementation of the instantiation and provisioning operations.

Each slice is ultimately controlled by and accountable to a *slice authority* (SA), which exports a well-defined, remotely accessible interface. The slice authority interface defines the operations used to create and manage slices, and to create the credentials that allow users to perform other actions on behalf of the created slice.

4 Authorization and Access Control

Authorization and access control is a core function of the SFA. Two primary classes of resources are protected by SFA-based authorization mechanisms. The first of these is slices, the containers for user-level experiments. The second is the collection of physical and logical resources implemented as, or made available by, components.

SFA’s authorization and access control architecture has the following goals.

- It is intended to be *semantically appropriate* for the management of slice-based network facilities.
- It is intended to make its trust model and decisions *explicit*. All decisions, delegations, and similar actions are enabled and managed by explicit credentials, rather than being side-effects of implicit naming relationships, user properties, and so on.
- It is intended to be *minimally communicative*. Communication among entities in the system to support authorization and access control should occur only where and when semantically required to perform the function. This property, together with the use of explicit credentials, is intended to support disconnected operation of different system elements to the maximum extent possible. [*Issue: ok, there’s probably a better way to say this..*]

4.1 Slices

The creation of a slice – essentially the assignment of a slice ID and creation of the initial credentials that enable further operations on, and on behalf of, the slice – is performed by a Slice Authority (SA), based on a request from one or more users. In so doing, the SA is agreeing that it is prepared to be accountable for the actions of the slice: it is certifying its trust in the slice’s designated Users.

The Slice Authority thus represents a point of connection between the technical authorization and access control mechanisms of the SF architecture and the social structure of trust management among users of SFA-based facilities. While some slice authorities may be entirely independent, it is desirable for the SFA to support more sophisticated social trust models common to its intended user community as well.

The current SFA design implements a trust hierarchy-delegation authorization model for slice authorities. It implements a SA hierarchy rooted at one or more top-level SAs. Below the top-level SAs, intermediate nodes in the hierarchy represent intermediate slice authorities, while the leaf nodes represent the slices themselves. Before creating a sub-node of its portion of the hierarchy, a slice authority must use an off-line process to verify that it is willing to vouch for the sub-authority or slice. It is assumed that each SA has been delegated authorization authority over its portion of the hierarchy, and need not refer slice or sub-authority creation requests to a higher level authority.

Note that other trust establishment and accountability models are possible – e.g., web of trust – but for simplicity, this document limits its discussion to this hierarchy-delegation model. We expect alternative models will be developed as realizations of the SF architecture become more widespread.

For fail-safe reasons, a slice registration has a finite lifetime, and it is necessary to periodically refresh the registration. This lifetime is reflected in the credentials created by the slice authority.

4.2 Components

Physical resources are encapsulated as components. Each component has an associated management authority, which represents the component's owner. The MA is responsible for defining and implementing the authorization policy governing use of the component. Tickets are used to implement this model, where a ticket represents a principal's right to (1) create a sliver (perhaps with some initial resources bound to it) on a component, (2) bind component resources to an existing slice, and (3) control a running slice. All three rights can be delegated. A ticket is essentially an RSpec signed by a component's MA, granting rights to allocate resources on one or more components. The holder of the ticket must go back to the component to split it into two tickets – but the expectation is that other tickets can be delegated by the slice identified in the ticket.

[Issue: above is unchanged from previous version, but will need attention depending on how the MA issue turns out.]

4.3 Flow

[Section intended to tersely describe timeline / flow of operations to implement A&A model. Problem: not quite enough context defined yet if this section stays here.]

[Issue: Articulate model of simple privileges and complexity of constraints that can be expressed in rspecs.]

5 Names & Identifiers

The SFA defines unambiguous identifiers – called *Global Identifiers* (GID) – for the set of objects that make up the federated system. GIDs form the basis for a correct and secure system, such that an entity that possesses a GID is able to confirm that the GID was issued in accordance with the SFA and has not been forged, and to authenticate that the object claiming to correspond to the GID is the one to which the GID was actually issued.

Specifically, a GID is represented as an X.509 certificate [X509, RFC-3280] that binds a Universally Unique Identifier (UUID) [X667] to a public key. The object identified by the GID holds the private key, thereby forming the basis for authentication. The authority that created and controls the object signs the corresponding GID X.509 certificate; this authority must be identified by its own GID. Any entity may verify a GID via cryptographic keys that lead back, possibly in a chain, to a well-known root or roots.

We also expect an SFA-based system to provide one or more registries that map information about objects – including human-readable names – to their GID. Section 8 describes a simple hierarchical registry that might be used to “bootstrap” an SFA-based system.

[Issue: above is somewhat muddled about whether requirement for GID is only self-certification, or PKI-like certification by roots. If naming is decoupled from trust (as above) and trust is reflected in explicit creds, then need to think through what purpose of hierarchical certification is – good chance not needed at all. Eliminate PKI assumption?]

6 Data Types

The SFA defines three key data types in addition to GIDs.

6.1 RSpec

A *resource specification*, or RSpec, describes a component in terms of the resources it possesses and constraints and dependencies on the allocation of those resources. Each resource (such as a processor or a network interface) is identified by an identifier (RID), which is assigned by the component manager and must be unique within the component. Thus, a combination of a component name and an RID is sufficient to unambiguously identify any resource in the federated system.

For example, an RSpec might describe a component’s processing capabilities (such as processor architecture and speed), its network interfaces (including bandwidth and the like), and privileged operations that can be invoked on the component (such as access to instrumentation, protected kernel state, and hardware accelerators). The purpose of the RSpec standard is to give component managers, user services, and end users a common resource vocabulary.

Most users are not expected to use RSpecs directly. In this sense, it can be thought of as a “machine language,” used below the user-visible level. It should concentrate on clarity of definition and expressiveness, with lower priority given to user-friendliness. As with any assembly or machine language, some users may choose to use this language directly, but we expect that most will use some high-level interface. Such high-level specifications will be “compiled” to an RSpec by the service that offers it. The SFA will support a wide variety of such

higher-level embedding services [GDD-07-44]. The ability for higher-level services to provide a user-friendly interface relies on RSpec syntax and semantics having the desirable properties of an output language for machine-compiled high-level specifications, so that it will be a suitable target for a range of front-ends.

Much of the complexity of the SFA is embedded in resource specifications. RSpecs are an extensible part of the SFA interface. As new resources and capabilities are added, these specifications will inevitably need to be extended. This will occur through the combination of a standardization process for extending RSpecs *and* appropriate technical design of the RSpec data structure, including hierarchical element name space and inheritance. This will allow new communities that federate to extend RSpecs within their own partition of the name space, and components that offer specialized resources will similarly extend the RSpec in their own name space.

[Issue: text below commented out while slice-RSpec questions get resolved.

“Note that an RSpec can correspond to either a sliver on a single component, or the set of slivers that span an entire slice. Whether we mean sliver-RSpec or slice-RSpec will be clear from the context.”]

6.2 Ticket

A component signs an RSpec – one that includes the right to allocate the corresponding resources – to produce a *ticket*. Such tickets are essentially credentials that are “issued” by a component, and later “redeemed” to acquire resources on the component. Tickets may also be delegated from one principal to another.

The SFA defines the format for tickets, and the set of rules governing their delegation. A ticket includes the following information:

Ticket = (RSpec, GID, StartTime, Duration, Privileges, Delegate, TicketID)

where **RSpec** describes the resources for which rights are being granted by the component; **GID** identifies the slice to which rights to allocate the resources are being granted; **StartTime** and **Duration** indicate the time period for which the ticket is valid; **Privileges** identifies the operations the holder is allowed to invoke on the CM; **Delegate** indicates whether the holder can delegate the ticket to another principal; and **TicketID** is a unique sequence number used by the component to ensure that the ticket is used only once. This information is signed by component that issues the ticket, and similarly re-signed when delegated.

Note that while all rights over component resources originate with the component itself – according to the policy defined by the owner – in practice, a component owner may implicitly delegate responsibility for a component to a management authority, in which case that MA signs the original ticket on behalf of the owner. In this case, we still think of the owner as defining the resource allocation policy for the component, but the MA effectively implements that policy by ensuring that the component runs software (in particular, a CM) that enforces that policy. To be more specific, the architecture treats a management authority as the principal that speaks for a component; owners that manage their own components are effectively the MA for those components.

[Issue: leaving aside semantics, how does above work pragmatically? Seems to require that the MA also implement the CM interface for all managed components, or how does it get the request to sign the ticket? Yes, it does imply this. -- llp]

Tickets can grant two privileges – **instantiate** and **bind** – indicating that the holder has the right to create a sliver on a component with the specified set of resources, and bind the specified set of component resources to an existing sliver, respectively. The specific operations corresponding to these privileges are defined in Section 7 .

[Issue: text below commented out while slice-RSpec questions get resolved.

“In as much as RSpecs can be either sliver-specific or slice-wide, a ticket can also correspond to a single sliver or an entire slice. The case should be clear from the context.”]

6.3 Slice Credential

A slice credential carries the right to create and manipulate slivers belonging to a slice in a set of willing components for the period of time during which the slice is said to be live, the right to refresh that live-ness period, and the right to manipulate the slice itself by modifying the list of bound users. It is given by the tuple:

SliceCredential = (GID, LifeTime, Privileges, Delegate)

where **GID** identifies the slice or slice authority to which rights to allocate the resources are being granted; **LifeTime** indicates the time period for which the slice is valid; **Privileges** identifies the operations the holder is allowed to invoke on behalf of the slice; and **Delegate** indicates whether the holder can delegate the credential to another principal. This information is signed by the SA responsible for the slice, and similarly re-signed when delegated.

Slice credentials can grant four privileges – **modify**, **embed**, **refresh**, and **control** – indicating that the caller has the right to modify the list of users bound to the slice, embed the slice in the substrate (or acquire additional resources for an existing slice), refresh the life time for an existing slice, and control a running slice, respectively. Note that the **embed** privilege indicates that the SA has taken responsibility for the slice; one or more components must accept and grant resources to the slice as a separate action. The specific operations corresponding to these privileges are defined in Section 7.

7 Interfaces

The following describes, in high-level terms, the interfaces provided by the core set of SFA objects. A candidate set of concrete interfaces is defined elsewhere. The reader should keep in mind that while these interfaces are distinct, a given realization is free to support objects that export multiple of these interfaces.

[TODO: Make all arguments explicit.]

7.1 Slice Interface

A slice is created by contacting an appropriate slice authority with a creation request:

`SliceCredential = CreateSlice(Info, GID)`

where `GID` identifies the principal creating the slice, and `Info` is information about desired slice lifetime, users, and so on.

[Issue: Is GID defined to be a user? Does this user end up as the initial user bound to the slice? If not, Info must include at least one user GID to bind, so that the next operation will be usable..]

Issue: this is violating the rule of explicit credentials – assumes the SA knows its users. Got to stop the recursion somewhere, but can consider whether this is the right approach..

The returned `SliceCredential` gives the right to bind users to the slice and otherwise modify the slice authority's information about the slice.

Once a slice has been created by a trusted slice authority, any user bound to the slice can invoke the following operation on the slice (specifically, the SA for the slice) to retrieve a credential for the slice:

`SliceCredential = GetCredential(GID)`

This credential is signed by the sponsoring SA, where embedding the slice in a set of components requires `SliceCredential` include the `embed` privilege; updating a slice's `LifeTime` value requires the `refresh` privilege; and controlling (starting, stopping, deleting) an embedded slice requires the `control` privilege.

[Issue: earlier text described use of a SliceCredential to control modification of the slice, but didn't mention use of credential to control issuance of these new SliceCredentials. That's consistent with this interface, but not with a fully credential-based design.]

[Issue: does a slice with the modify privilege do so via some registry interface, or does the slice interface include an UpdateSlice operation.]

Note that slice authority may also support a set of operations to embed, provision, and control the slices it authorizes, implying that the SA is exporting both the slice interface and the component interface described in the next section. We return to the question of how a given implementation might offer various combinations of interfaces in a later section.

7.2 Component Interface

Once a user possesses a slice credential, it can invoke the following operations on a component (specifically, the CM for the component) to instantiate, provision, and control the slivers that make up the slice. Keep in mind that single component is able to create only local slivers, meaning that the following operations must be invoked on each component that the slice is expected to span. This slice-wide activity can be done directly by a user, or indirectly through either a slice manager acting on behalf of set of slices or through an aggregate acting on behalf of a set of components. Section 9 discusses these possibilities in more detail.

Although the syntax of the interface is identical when applied at the sliver level and the slice level, the failure semantics are not. This is because applying the operations at the slice level may or may not succeed when each individual components belonging to the slice is contacted; a

given component may simply be unreachable at that moment. That is, the slice-level variants of these operations must be considered “best effort.” Fortunately, both tickets and slice credentials time out, bounding the negative consequences of any such failures.

7.2.1 Embedding a Slice

A user invokes a

```
Ticket = GetTicket(SliceCredential, RSpec)
```

operation on a component to acquire rights to component resources. The returned ticket effectively binds the slice to the right to allocate on the component the requested resources/rights. Whether or not the call succeeds depends on any access control implemented by the component, the local resources available on the component, and the allocation policy implemented by the component (on behalf of the component owner). The `SliceCredential` identifies the slice and indicates the period of time for which the slice’s registration is valid; the component likely bounds the returned ticket’s duration accordingly. `SliceCredential` must also include the `embed` privilege.

An entity that holds a ticket may also split off a portion of the corresponding resources – effectively creating a new ticket – with a

```
Tickets[ ] = SplitTicket(Ticket, RSpec)
```

operation; other privileges are preserved. Note that splitting a ticket requires calling the component that originally issued the ticket, independent of how many times the ticket has been delegated.

Once a principal possesses a ticket, it can create a sliver on the component and bind new resources to an existing sliver by invoking an

```
RedeemTicket(Ticket)
```

operation. Creating a new sliver requires the `instantiate` privilege and augmenting an existing sliver with additional resources requires the `bind` privilege.

As a convenience, a component may support a

```
EmbedSlice(SliceCredential, RSpec)
```

operation, which effectively combines `GetTicket` and `RedeemTicket` in a single call.

7.2.2 Controlling a Slice

Component managers also support three control operations:

```
StopSlice(SliceCredential)
```

```
StartSlice(SliceCredential)
```

```
DeleteSlice(SliceCredential)
```

where the `SliceCredential` parameter passed to all three operations identifies the slice being controlled, and must include the `control` privilege. The first two operations stop and start the

execution of an existing slice. The slice retains any acquired resources on the component, although a component that uses work-conserving schedulers is free to utilize those resources for the duration of the suspension.

Note that there is *no* assumption that user-level sliver state is preserved across a `StopSlice()` `StartSlice()` pair. These two operations are often used in tandem to “reboot” a slice, but are offered as a pair to give a user (or a helper service managing the slice) an opportunity to “clean up state” before restarting. The final operation removes the slice from the component and releases all of its resources.

Note that while starting and starting a slice is best viewed as a reboot, the pair can also effectively suspend/resume the slice, if coupled with a high-level service that preserves and restores the slice’s internal state. For this to work, the high-level service must be notified that a `StopSlice()` has occurred. The architecture leaves it to each execution environment to define how (if at all) it notifies a slice that it is about to be stopped (and has restarted) on a given component.

7.2.3 Slice Information

Components also support three informational operations:

`Slices[] = ListSlices(GID)`

`RSpec = GetResources(GID)`

`GID = GetSlice(Signature, GID)`

They are used to learn the GIDs of the set of slices instantiated on that component, the set of resources available on the component, and the GID of a slice responsible for sending a packet with a given signature, respectively. The GID parameter passed to all three operations identifies the caller. There are (currently) no restrictions on who may call these operations.

The `GetSlice` call can only be invoked relative to a packet to/from a component and the legacy Internet. This call is meaningful on only those components that are connected to the legacy Internet, where the `Signature` parameter is a [protocol, source IP, destination IP, source port, destination port] five-tuple.

7.3 Management Interface

A management interface is used to boot and configure components, bringing them into a state that they can support the component interface to create and control slivers. The interface is also used to bring the component into a safe state should the component be compromised. Both individual components and aggregates representing a set of components can be expected to support the management interface.

The management interface includes five operations:

`SetBootState(GID, State)`

`State = GetBootState(GID)`

`Reboot(GID)`

`Components[] = ListComponents(GID)`

Status = GetStatus(GID)

The first operation is used to set the boot state of a component to either **debug** or **production**, the former implying the component is in a safe state with all slice-related operations disabled. The second operation is used to learn a component's boot state and the third operation forces the component to reboot into the current boot state. When applied to an aggregate, these operations affect only the identified component, not all components affiliated with the aggregate.

The last two operations return information about component(s): **ListComponents** returns the GID for each of a set of components (this set contains a single element when applied to an individual component) and **GetStatus** returns component-specific status information (e.g., uptime, current load).

Note that we expect a given component (or aggregate) to support a much richer set of management-related operations, effectively extending the required operations listed here. The management interface defines only the minimal set of operations **all** components (aggregates) must support.

[TODO: Need to talk about what principals are allowed to invoke what management ops, and how to delegate.]

8 Name Registry

A *name registry* maps strings to GIDs, as well as to other domain-specific information about the corresponding object, such as the URI at which the object's manager can be reached, an IP or hardware address for the machine on which the object is implemented, the name and postal address of the organization that hosts the object, and so on. While one or more functioning name registries are an important part of any practical slice-based facility, such registries should be viewed as a useful building block of the system, not part of the core definition, per se. This section outlines what a "bootstrap" name registry might look like.

8.1 Example Registries

Although many different types of registries might exist in an SFA-based system, we initially define two default registries—a *component registry* and a *slice registry*—both of which define a hierarchical name space corresponding to a hierarchy of authorities that have delegated the right to create and name component and slice objects, respectively. These registries assume a top-level naming authority trusted by all entities, resulting in names of the form:

top-level_authority.sub_authority.sub_authority.name

For example, "geni" and "planetlab" might be top-level authorities; it is possible that other similar authorities might federate in accordance with the SFA. This is not to imply that all federation is strictly among and other top-level authorities, since even in the context of a single top-level authority, we allow for multiple autonomous MAs that agree to federate their resources.

The component registry maintains information about a hierarchy of *management authorities*, along with the set of components for which the MAs are responsible. This registry binds a human-readable name for components and MAs to a GID, along with a record of information

that includes the URI at which the component's manager can be accessed; other attributes and identifiers that might commonly be associated with a component (e.g., hardware addresses, IP addresses, DNS names); and in the case of an MA, contact information for the organization and operators responsible for the set of components. For example,

`geni.us.backbone.nyc`

might name a component at the NYC PoP of GENI's US backbone. In this case, the `geni.us.backbone` management authority is responsible for the operational stability of the set of components in the backbone network.

The slice registry maintains information about a hierarchy of *slice authorities*, along with the set of slices for which the SAs have taken responsibility. This registry binds a human-readable name for slices and SAs to a GID, along with a record of information that includes email addresses, contact information, and public keys for the set of users associated with the slice; and in the case of an SA, contact information for the organization and people responsible for the set of slices. For example,

`planetlab.eu.inria.dali`

might name a slice created by the PlanetLab slice authority, which has delegated to the EU, and then to INRIA, the right to approve slices for individual projects (experiments), such as Dali. PlanetLab defines a set of expectations for all slices it approves, and directly or indirectly vets the users assigned to those slices.

Note that both the GENI and PlanetLab management authorities are expected to maintain an operational set of components capable of hosting experiments, and their respective slice authorities are expected to support slice creation on behalf of network and distributed systems researchers. Because it is possible that other related facilities will federate with GENI and PlanetLab, and there will be other uses of the greater federated system, we allow for the possibility that other top-level slice authorities may support other policies and purposes. For example, there could exist a top-level slice authority that permits slices running for-profit services.

Also note that the registries may be distributed, where a server that implements one portion of the hierarchy includes a pointer (URI) to a server that implements a sub-tree of the hierarchy. We expect slice and management authorities will often implement a registry server for the sub-tree of the hierarchy for which they are responsible.

8.2 Registry Interface

A registry records facts about the objects in the systems (e.g., components and slices), and the principals (e.g., MAs and SAs) that are responsible for them. Each entry in the registry is of the form:

Record = (Name, GID, Type, Info)

where

Type = Authority | Component | Slice | User

and

Info = (LifeTime, PublicKey, PostalAddress, Phone, Email), if Type = Authority

Info = (LifeTime, PublicKey, PostalAddress, Phone, Email), if Type = User

Info = (LifeTime, PublicKey, URI, LatLong, IP, DNS), if Type = Component

Info = (LifeTime, PublicKey, URI, User1,... UserN), if Type = Slice

The registry interface supports the following five operations:

GID = Register(Name, Type, Info, GID)

Remove(Name, GID)

Update(Name, Info, GID)

Record = Resolve(Name)

Record[] = List(Name)

The first two operations are used to register and un-register objects and principals. The third operation is used to update information about an entry. Each record includes live-ness information (denoted **LifeTime**), which must be periodically refreshed (using **Update**) or it is automatically removed. The fourth operation is used to learn the information bound to a given **Name** and the final operation is used to retrieve information about the set of objects managed by a given authority.

The **GID** parameter in the first three operations identifies the caller. This must be the responsible authority in the case of **Register** (the returned **GID** is for the newly registered object or authority), and may be either the responsible authority or the object itself in the case of **Remove** and **Update**. Any principal may call the last two operations.

The **URI** field in the **Slice** and **Component** registry entries identify the slice authority and component manager, respectively, for the corresponding object.

Note that all the information recorded in a registry is relatively static, meaning we expect the **Update** operation to be relatively rare. Users that need up-to-date information about dynamically changing attributes should query the relevant object directly, for example using the **GetResources** call described in the next section.

9 Implementation Options

While the four interfaces defined in the previous two sections are distinct, a given system might conflate them in various ways. This section outlines three possible examples.

First, a slice authority might implement both the registry interface (recording information about a set of slices) and the slice interface (granting credentials for those slices). This slice authority might even support the component interface, which the users of those slices could invoke to create slices across a set of components. Should the slice manager include the component interface, it would need to acquire tickets for component resources since it does not manage any components itself.

Second, an aggregate – representing a set of components and running on behalf of a management authority that is responsible for those components – might support the registry

interface (recording information about those components), the component interface (allocating resources and creating slices across the set of components), and the management interface (providing a means to manage the constituent components). In this case, users would need to present the aggregate with a slice credential acquired from some slice authority in order to create a slice.

As a final example, PlanetLab combines all four interfaces into a single program, called PlanetLab Central (PLC). That is, PLC implements both a slice registry and a component registry, supports a slice manager that is used to create slices across PlanetLab's components, and exports a management interface that is used to manage PlanetLab's components. It is also worth noting that PlanetLab's full management interface includes many more operations than the minimal set defined earlier in this section, allowing the operations staff to manage PlanetLab components in a PlanetLab-specific way.